Lecture Notes for CSCI 351: Distributed Computing
Set 5-Clocks

Professor Talmage

March 27, 2023

---

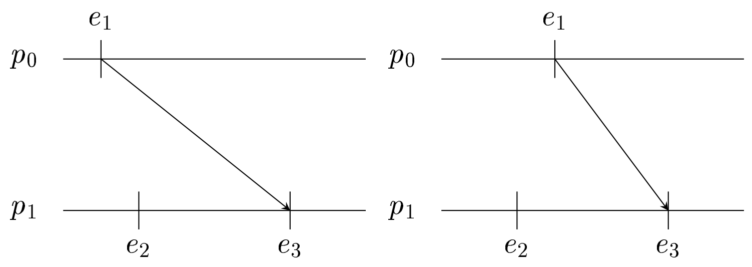
1 The Problem

In a sequential system, time is simple: we can model it by an increasing sequence of integers. It is easy to compare two events and determine which happened first: The event with smaller clock time happened first. In a distributed system, time is not so simple, since it may not be possible to know the exact time when an event happened at a remote process, but that event may still influence local behavior. We want a way to define timestamps for events in a distributed system that respects the lack of knowledge of when events actually occur, but allows us to see the relationships between events at different processes.

We will start by considering an asynchronous, message-passing system.

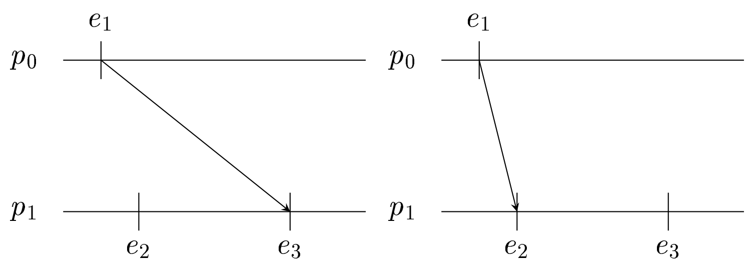
2 Happens Before

First, we need to better understand what information our timestamps will actually capture. In an asynchronous system, there is no way to learn exactly when something happened at a remote process, as its clock can run at any rate and messages can be arbitrarily delayed. Consider the following two timelines, in which we draw a timeline (graphed against real time, increasing to the right) for the events at each process. Remember that processes themselves have no knowledge of the passage of real time.



In the first execution, e_1 precedes e_2 in real time, while in the second, e_2 precedes e_1. But since the only message from p_0 to p_1 arrives at the same time in both executions, p_1 cannot tell that there is any difference. Nor can p_0, since any local clock it has may have the same value at e_1 in both executions. Thus, the two executions are indistinguishable, since they are similar to all processes.

Now consider the following two executions:



Here, e_1 happens at the same real time in both executions, but the message from p_0 to p_1 arrives sooner. This may change p_1's behavior between the two executions, so this is something we want to capture with our timestamps. This is a *causal* relationship, meaning that the difference may cause some difference in the executions.

Exercise: Try to draw all scenarios in which we can conclude that one event can causally affect another. *(Hint: There are three rules, which may appear in more scenarios.)*

Definition 1. In an execution E, we say that event e_1 *happens before* event e_2, written as $e_1 \prec e_2$, if one of the following is true:

1. e_1 and e_2 are events at the same process and e_1 occurs before e_2.
2. e_1 is the send event for message m and e_2 is the receive event for message m.
3. There exists an event e such that $e_1 \prec e$ and $e \prec e_2$ (*Transitive Closure*)

Exercise: Try to draw an execution in which one event causally influences another, but the relationship is not captured by this definition. Convince yourself this is impossible.

This definition captures all possible causal relationships, since in an asynchronous system, timing carries no meaning.

3 Logical Clocks

The happens-before relation is all well and good, but it will be far more useful if processes know the relationship between events than just as an analysis tool. We defined it based on a total view of the system, which processes do not have. How can processes track and understand happens-before?

3.1 Idea 1: Lamport's Logical Clocks

Each process tracks the current time as an integer. This is known as a *logical clock*, since it bears no direct relationship to the physical notion of time.

- Every time a process performs a computation event, it increases its clock by one, and the new time is the *timestamp* of that event.
- When sending messages, include the clock value of the send event as a timestamp.
- In a receive event, the (receiving) process increases its local clock to one more than the maximum of its local clock value and the message's timestamp.

Exercise: Does this algorithm yield timestamps which capture the happens-before relation, or are they stronger or weaker?

We can then use the integer $<$ order to compare timestamps and conclude the following relationships:

1. If e_1 and e_2 are events at the same process, $ts(e_1) < ts(e_2)$ iff $e_1 \prec e_2$.
2. If e_1 is $send(m)$ and e_2 is $receive(m)$, then $ts(e_1) < ts(e2)$.

Theorem 1. *If e_1 and e_2 are events in execution E and $e_1 \prec e_2$, then $ts(e_1) < ts(e2)$.*

Proof as already discussed, plus dealing with transitivity, which follows from the transitivity of $<$ on integers.

This does **not** equal the happens-before relation. Note that the reverse implication does not hold. For example, in the first execution we drew, $ts(e_1)$ could be 1 and $ts(e_2)$ could be 10, but $e_1 \not\prec e_2$.

These logical clocks, known as Lamport's Logical Clocks after Leslie Lamport, who is one of the pioneers of distributed computing and a Turing Award winner, are sufficient for some tasks. You're using them in Project 2. But it would nice to go the other direction, comparing timestamps to determine whether one event happened before another.

3.2 Vector Clocks

The idea here is that, since different processes' logical clocks may move at very different rates, we want to not just jump forward to the largest every time we receive an update. Instead, we want to try to keep a view of where each process is. Thus, each process' local clock will be a vector (list), with one value for each process in the system.[1][2] Each process will track the latest value of which it is aware for each process' local logical clock.

- To compare two timestamps, we use the partial order $(c_0, c_1, \ldots, c_{n-1}) \leq (d_0, d_1, \ldots d_{n-1})$ iff $c_i \leq d_i$ for all $0 \leq i < n$.
- $C < D$ (C and D are entire vectors/lists) if $C \leq D$ and $C \neq D$.

  Exercise: Give 3 pairs of smaller and larger vectors of length 4 (try to make them interesting), and one pair which cannot be compared.

  - This is what we mean by *partial order*–we cannot always compare to elements.
- Process i increments the ith component of its clock vector in every local computation step.
- When a process sends a message, it attaches its entire clock vector.
- When a process receives a message containing clock vector D, it updates each component i of its local clock to the larger of C_i and D_i, where C is the local clock vector. (Recall that it also increments its own component in this computation event.)

  Exercise: Argue that no process will ever need to increase its own component (beyond the receive event's increment) based on a clock vector it receives in a message.
- An event's timestamp is the local clock vector after it is updated.

Exercise: Does this clock algorithm capture the happens-before relation? That is, if e happened before f, is its timestamp smaller, and if $ts(e) < ts(f)$, did e happen before f?

Definition 2. We say that events e_1 and e_2 are *concurrent* if neither $e \prec f$ nor $f \prec e$.

[1]This is still a logical clock, since it's not based on the physical notion of time.
[2]No, such an algorithm cannot be uniform.

- Note that this is a formal usage of the term, and is more restrictive than the informal, intuitive way we have used "concurrent" so far.
- We will actually have another, different formal definition of concurrent later when we talk more about implementing shared data structures.

By cases:

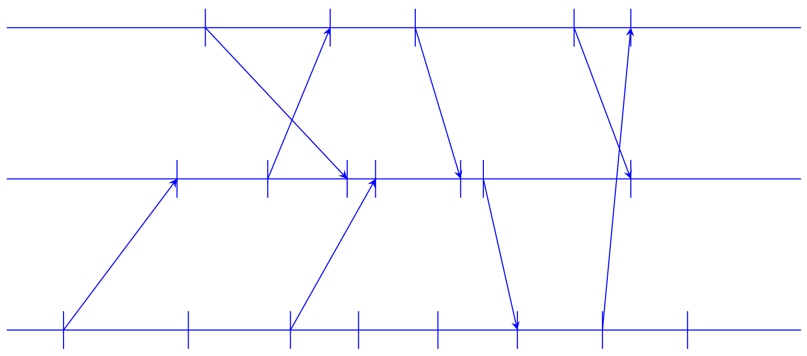
1. As before, if e and f are at the same process, $ts(e) < ts(f) \Leftrightarrow e \prec f$.
2. If $e = send(m)$ and $f = receive(m)$, then the receiving process updates its vector clock so that every entry is at least as large as the vector send in e, so $ts(f) > ts(e)$.
3. Transitivity is less trivial, but still holds by the transitivity of integer $<$.

Exercise: Work through the proof of transitive closure.

4. Assume e and f are concurrent events at p_i and p_j, respectively. Suppose the component of $ts(e)$ for p_i is t. Then the component of $ts(f)$ for p_i must be less than t, because no message chain has gone from p_i to p_j to update that value. Similarly, the component for p_j in $ts(f)$ is large than the component for p_j in $ts(e)$. Thus, neither $ts(e) < ts(f)$ nor $ts(f) < ts(e)$.

Theorem 2. *Using vector clocks, if e and f are any two events in an execution, then $ts(e) < ts(f) \Leftrightarrow e \prec f$.*

Exercise: In the following timeline, label each event with its Lamport-clock timestamp. Repeat with vector clock timestamps.



4 Clock Synchronization

We want to now move to more practical applications. Real computers have clock hardware, typically quartz oscillators.[3] To do this, we will add clocks, and some associated assumptions, to our model, and try to make those clocks useful by synchronizing them.

Assume that every process p_i has a clock component, called a *hardware clock* to indicate that it is a signal source external to the algorithm and denoted HC_i. The process can read this clock and use the value seen to determine its next action (input to the transition function). When read, a hardware clock returns a function of real time: $HC_i(t)$, though recall that the process has no direct access to real time t.

[3]Fun aside–you can buy a rack-mount atomic clock and connect it to a computer that needs a really precise clock. A quick internet search finds this: `https://technical-sys.com/shop/quartzlock/a1000a-rubidium-atomic-master-audio-clock/`, which claims errors on the scale of fractions of a part per billion in a year. A similarly arbitrarily-searched source `https://freqelec.com/quartz-master-clocks/` offers quartz clocks with precision 1/1000 as good over 24 hours.

- Hardware clocks must be strictly increasing functions, but may be any such (speed up, slow down, jump, etc.)
- To start, we'll assume for sanity and to make the problem tractable that hardware clocks run at the same rate as real time, but may be ahead or behind. Then we can say that $HC_i(t) = t + c_i$, where c_i is a constant value known as *offset*.
- Processes can construct an *adjusted clock* AC_i by setting an adjustment variable adj_i, s.t. $AC_i(t) = HC_i(t) + adj_i = t + c_i + adj_i$.
- Ideally, $adj_i = -c_i$, so that the adjusted clock returns real time. As we will see, that is generally not possible.
- You might next think that we want to minimize $|AC_i(t) - t|$. But what is more important is the difference between the clocks of different processes. Besides, we don't have a good way to know whether we have gotten AC_i near t, since there is no reliable way to read t.

Definition 3. An algorithm A achieves ϵ-synchronized clocks if it terminates in a finite time and after termination, $|AC_j(t) - AC_i(t)| \leq \epsilon$, for any processes p_i and p_j.

- ϵ is called the *clock skew*, or simply skew.
- Skew bounds how different two processes' clocks can be.
- We compare clock algorithms by the skew they guarantee. We are less concerned with other measures, such as running time, since we will often assume that we have first run a clock-synchronization algorithm, then run whatever algorithm we need in a synchronized system.
  - This only works when hardware clocks always run at the same rate as real time. I know of results for hardware clocks that run at a fixed multiple of real time, but those cannot be run once and then set aside.

Exercise: What is our main impediment to writing a clock synchronization algorithm at this point?

We cannot synchronize clocks in a purely asynchronous system, since arbitrary message delays mean that the clock values processes send to each other are next to meaningless, as the recipient cannot know how old the value is. We will thus move to a stronger model:

- Message Passing
- Fault-free
- Partially synchronous: Every message takes at most d real time to arrive, and at least $d - u$. That is, d is the maximum message delay and u is the uncertainty in message delay.
  - We assume d and u are known to processes.
- Fully-connected: Every process can send a message directly to any other process.

4.1 2-Process Synchronization

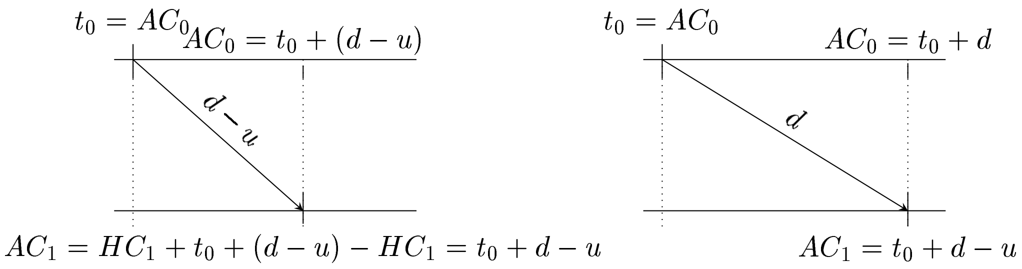
We start with a simple case. Suppose there are only two processes, and we want to minimize the skew between their clocks.

Exercise: Bearing in mind the range of possible message delays ($[d-u, d]$), try to come up with a procedure to get the two processes' adjusted clocks as close to each other as possible.

- The only useful way to exchange information is for (at least) one process to read its local clock and send that value to the other process. Assume, WLOG, that p_0 sends its local clock value, t_0.
- When p_1 receives p_0's local clock value, it needs to determine how much to adjust its clock to match p_0's as closely as possible. But it doesn't know how old the clock value it got from p_0 is, as it doesn't know how long the message took.

  1. If p_1 assumes the message was quick, then it thinks p_0's local clock is now at $t_0 + d - u$. We can set up an equation for the adjustment value p_1 should set: Recall that $AC_1 = HC_1 + adj_1$, and we want $AC_1 = t_0 + (d-u)$. Solve for adj_1: $adj_1 = t_0 + (d-u) - HC_1$.

     We now need to figure out the maximum possible skew. Consider the first timeline graph below, where the message actually took $d-u$ real time to arrive. In this case, the processes' adjusted clocks will match exactly.

     However, if the message was actually slow, taking d time to arrive, then the difference between adjusted clocks is u, as shown in the second figure above.
  2. If p_1 assumes the message was slow, taking d time, then we get a symmetric scenario, where the skew could be as much as u if the message was actually fast.
  3. To minimize the maximum difference between adjusted clocks, p_1 should assume that the message took $d - u/2$ time to arrive. Now, the two adjusted clocks will differ by at most $u/2$.
- Both processes do this at once

4.2 Shifting for Lower Bounds

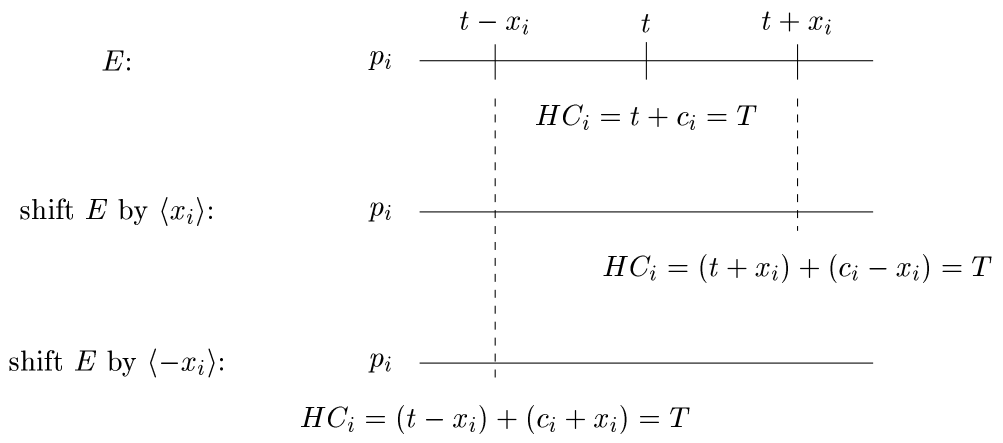
We want to show that our two-process clock synchronization algorithm is the best possible. That is, we want to show that any clock synchronization algorithm has an execution which results in a skew at least that large. To do this, we need to introduce a new tool, *shifting*. We can use this technique to show a variety of lower bounds for different distributed problems, as it relies on the uncertainty of timings and message delays to show that different executions, in which events happen in different real-time orders, are indistinguishable.

The general outline of a shifting proof is as follows:

1. Assume an algorithm performs better than the lower bound we want to show.
2. Build an execution of that algorithm.

3. Move all events at some process(es) in real time, but change the offsets of those process(es)' hardware clock(s) an equal but opposite amount. This builds another execution which is indistinguishable to the processes, as all events happen at the same local clock times, despite happening at different real times.

4. For skew bounds, note that changing hardware clocks changes adjusted clocks, which may change skew. If skew increases above the bound the algorithm claims to guarantee‘, we have a contradiction.

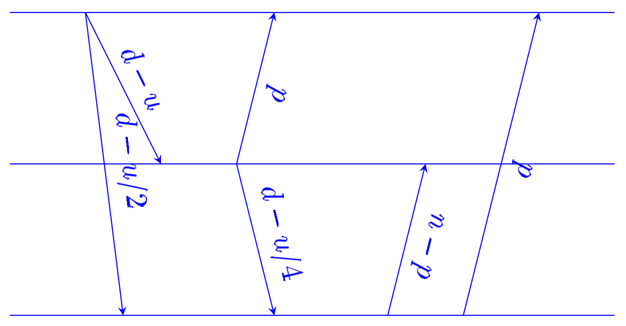

E:
p_i
$t - x_i$
t
$t + x_i$
$HC_i = t + c_i = T$
shift E by $\langle x_i \rangle$:
p_i
$HC_i = (t + x_i) + (c_i - x_i) = T$
shift E by $\langle -x_i \rangle$:
p_i
$HC_i = (t - x_i) + (c_i + x_i) = T$

Note that the signs on shifts can be a bit confusing. If we shift a process by x, that means we add x to the real time when each event occurs, which means that we then adjust the clock offset by $-x$ to make them appear to the process to happen at the same local time.

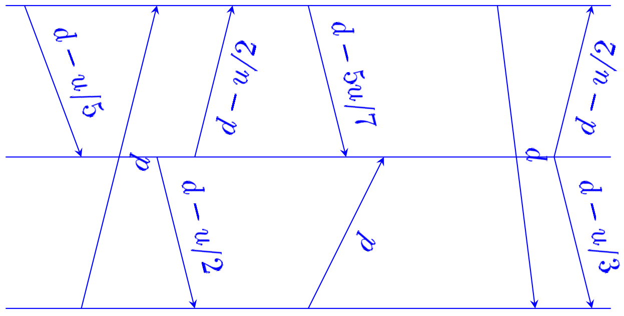
In general, we can shift each process by a different amount. Thus, we use the notation $shift(E, x)$, where E is an execution and $x = \langle x_0, \ldots, x_{n-1} \rangle$ is a vector of length n. We shift each process p_i by x_i.

- Each p_i's new hardware clock, $HC_i' = HC_i - x_i$
- Message delays are affected, too: Every message from p_i to p_j now has delay $D - x_i + x_j$, where D is the delay (real time from send to receive) of the message in E.
  - This is important, because this limits how far we can shift executions. We certainly don't want to make a message arrive before it was sent.
  - Further, we must respect the $[d - u, d]$ interval of admissible message delays. If the execution is not admissible, the algorithm isn't required to behave correctly. (Technically, any and all behavior is "correct" if the execution is inadmissible.)

Exercise: Shift the following execution by the vector $\langle -u/2, u/4, 0 \rangle$. Message arrows are labeled with their delays. Label each message in the shifted execution with its delays.



Exercise: Find a shift vector for the following execution that is **not** admissible, then find a non-zero shift vector that is admissible.



We can now use shifting to prove a lower bound on the skew between the clocks of two processes in our current model.

Theorem 3. *No algorithm can guarantee a clock skew less than $u/2$ in a 2-process system.*

Proof. Assume in contradiction that some algorithm A synchronizes clocks and provides a guaranteed skew $\epsilon < u/2$.

Let E be an admissible timed execution [4] of A in which all messages from p_0 to p_1 take $d - u$ time and all message from p_1 to p_0 take d time. Since these delays are allowed by our model, E is an admissible execution and A must provide the guaranteed skew.

Now, shift E to construct E', a new timed execution. We choose a specific shift vector: $E' = shift(E, \langle -u, 0\rangle)$. That is, all events at p_0 happen u earlier in real time, while events at p_1 do not move. We need to check that E' is admissible, or A does not have to work: By the formula above,

- messages from p_0 to p_1 now have delay $(d - u) - (-u) + 0 = d$ and
- messages from p_1 to p_0 now have delay $(d) - 0 + (-u) = d - u$.

These are within the required bounds, so E' is an admissible execution, and A must work correctly.

We can also conclude that $AC'_0 = AC_0 + u$, while p_1's adjusted clock hasn't changed. By A's guarantee, at any real time T after A terminates, we have $AC'_1(T) \geq AC'_0(T) - \epsilon$. Substituting back to adjusted clocks in E, we have $AC_1(T) \geq (AC_0(T) + u) - \epsilon$. But A also guarantees that $AC_0(T) \geq AC_1(T) - \epsilon$[5]. Substituting the first inequality into the second, we get

$$AC_0(T) \geq (AC_0(T) + u - \epsilon) - \epsilon$$

Reducing, we have $\epsilon \geq u/2$, directly contradicting the assumed performance of A. □

4.3 More Processes

We will now expand our results to systems of more than two processes. We will use similar ideas and techniques to bound the achievable skew.

Theorem 4. *For every algorithm guaranteeing ϵ-synchronized clocks in a complete communication network, $\epsilon \geq u(1 - 1/n)$.*

Proof. Let A be a clock-synchronization algorithm guaranteeing ϵ-synchronized clocks. Let E be an admissible timed execution of A where for any process IDs $i < j$, messages delays are

[4] A *timed execution* is an execution with a real time associated with each event.
[5] These bounds come from the bound on the absolute difference of any two processes' clocks.

- $d - u$ for all messages from p_i to p_j and
- d for all message from p_j to p_i.

That is, all message going to a higher ID are fast and all messages going to a lower ID are slow. We will construct a general shift to show that for any process $p_k, k > 0$, $AC_{k-1} \leq AC_k - u + \epsilon$. Once we have this relationship for all processes, we can deduce the desired bound.

Let p_k be any process other than p_0. Define $E'_k = shift(E, x)$, where

- $x_i = -u$ for $0 \leq i \leq k-1$
- $x_i = 0$ for $k \leq i \leq n-1$

Message delays in E'_k are now as follows, for any pair of process IDs $i < j$:

- If $j < k$ or $i \geq k$, then nothing changes and messages from p_i to p_j are delayed $d - u$, from p_j to p_i are delayed d.
- If $i < k$ and $j \geq k$, then messages from p_i to p_j have delay d and those from p_j to p_i have delay $d - u$.

Exercise: Draw a timeline for 3-4 processes with the given delays and messages to illustrate each case. Shift your execution and verify that these new delays are correct.

Since all message delays are in the interval $[d - u, d]$, E'_k is admissible and thus A must work correctly in E'_k. That is, it must synchronize all clocks within ϵ. Specifically, $AC'_{k-1} \leq AC'_k + \epsilon$. If we substitute back to the original adjusted clocks in E, we have

- $AC'_{k-1} = AC_{k-1} + u$, since p_{k-1} shifted earlier by u.
- $AC'_k = AC_k$, since p_k did not shift.

Thus, $AC_{k-1} = AC'_{k-1} - u \leq AC_k + \epsilon - u$. This inequality must hold for E, even if E'_k did not happen, because it could. Further, we get a similar inequality for any $0 < k \leq n-1$, since we could construct E'_k for each value of k, so we have an entire set of inequalities:

$$\begin{aligned} AC_0 &\leq AC_1 + \epsilon - u \\ AC_1 &\leq AC_2 + \epsilon - u \\ &\dots \\ AC_{n-2} &\leq AC_{n-1} + \epsilon - u \end{aligned}$$

We want to chain these together so that we can cancel out all AC terms and find a bound directly on ϵ, but we do not currently have a relationship between AC_0 and AC_{n-1}. But the basic guarantee of skew as the maximum difference between any two processes gives us a starting point. If we start with the skew bound, then substitute through all of the inequalities for different values of k, we obtain a single inequality we can solve for ϵ:

$$\begin{aligned} AC_{n-1} &\leq AC_0 + \epsilon \\ &\leq AC_1 + 2\epsilon - u \\ &\leq AC_2 + 3\epsilon - 2u \\ &\dots \\ &\leq AC_{n-1} + n\epsilon - (n-1)u \\ \epsilon &\geq u(1 - 1/n) \end{aligned}$$

□

What does this bound mean? As n increases, the $1/n$ term will decrease, so the skew will get closer and closer to u, the uncertainty in message delay. Thus, for synchronization, the uncertainty in message delay is more problematic than the delay.

Exercise: Can you give an intuitive explanation for why high maximum message delay but low (or zero) uncertainty is easier to work with than the reverse situation?

Exercise: What does this lower bound imply for a fully asynchronous system?

Upper Bound Now that we have a lower bound, we want to know whether it is tight. That is, is it possible to achieve $\epsilon = (1 - 1/n)u$, or is there a tighter (larger) lower bound we can prove?

Exercise: Discuss which you think is true, and try to justify your answer, by outlining either a way to prove a larger lower bound or an algorithm to achieve $\epsilon = (1 - 1/n)u$ skew.

Idea 1: Suppose that we designate one process as "center" (in other words, elect a leader). We can then have every other process run the 2-process synchronization algorithm with that center.

Exercise: What skew guarantee does this algorithm provide?

Idea 2: We need every process to simultaneously synchronize with all others. To do this,

1. Send your unadjusted clock to all processes.
2. Estimate the difference between your clock and each other clock.
3. Adjust your clock by the average of the estimated differences.

Each process is effectively attempting to calculate the average of all the clocks and adjust to match that average.

Exercise: Write pseudocode for this algorithm and attempt to analyze its skew.

Algorithm 1 Clock Synchronization Algorithm, code for p_i

```
Upon event: INITIALLY:
    send $HC_i$ to all
end Upon event:
Upon event: RECEIVE CLOCK VALUE $t_j$ FROM $p_j$:
    $diff_i[j] = t_j + d - u/2 - HC_i$
    if have received $n-1$ clock values then
        $adj_i = \left(\frac{1}{n}\right) \sum_{k=0}^{n-1} diff_i[k]$
    end if
end Upon event:
```

Theorem 5. *Algorithm 1 achieves optimal $\epsilon = (1 - 1/n)u$ skew.*

Proof. As for the 2-process algorithm, when p_i calculates $diff_i[j]$, its error is at most $u/2$. That is, $diff_i[j] = HC_j - HC_i + err_{ji}$, where err_{ji} is a constant with absolute value less than or equal to $u/2$: $-u/2 \leq err_{ji} \leq u/2$.

We can now algebraically determine the possible difference between the adjusted clocks of any two processes p_i and p_j:

$$|AC_i - AC_j| = \left| HC_i + (1/n)\sum_{k=0}^{n-1} diff_i[k] - HC_j - (1/n)\sum_{k=0}^{n-1} diff_j[k] \right|$$

Multiply HC_i and HC_j by n to incorporate in the summation.

$$= \frac{1}{n}\left|\sum_{k=0}^{n-1}(HC_i - HC_j + diff_i[k] - diff_j[k])\right|$$

Pull out the terms for p_i and p_j.

$$= \frac{1}{n}|(HC_i - HC_j + diff_i[i] - diff_j[i]) + (HC_i - HC_j + diff_i[j] - diff_j[j])$$
$$+ \sum_{k=0,k\neq i,j}^{n-1}(HC_i - HC_j + diff_i[k] - diff_j[k])\,|$$

Note that $diff_i[i] = diff_j[j] = 0$.

$$\leq \frac{1}{n}\left(|HC_i - HC_j - diff_j[i]| + |HC_i - HC_j + diff_i[j]|\right)$$
$$+ \frac{1}{n}\left(\sum_{k=0,k\neq i,j}^{n-1}|HC_i - HC_j + diff_i[k] - diff_j[k]|\right)$$

Substitute the definition of $err_{ji} = HC_i - HC_j + diff_i[j]$.

$$\leq \frac{1}{n}\left(|err_{ij}| + |err_{ji}|\right)$$
$$+ \frac{1}{n}\left(\sum_{k=0,k\neq i,j}^{n-1}|(HC_i - HC_j) + (HC_k - HC_i + err_{ki}) - (HC_k - HC_j + err_{kj})|\right)$$

Everything but the errors cancels.

$$\leq \frac{1}{n}\left(|err_{ij}| + |err_{ji}| + \sum_{k\neq i,j}|err_{ki} - err_{kj}|\right)$$

Each error has absolute value no more than $u/2$, so their difference has absolute value no more than u, and we obtain

$$|AC_i - AC_j| \leq (1/n)(u/2 + u/2 + (u(n-2))) = u(1 - 1/n)$$

□